C# Code Guideline

NHẤT NGHỆ

Hien Luong | ASP.NET Core | 2018

Contents

1 Introduction

This document describes rules and recommendations for developing applications and class libraries using the C# language. The goal is to guidelines to enforce consistent style and formatting and help developers avoid common pitfalls and mistakes.

Specifically, this document covers ***Naming Conventions, Code Style***.

2 Naming Conventions and Standards

Note *:*

The terms Pascal Casing and Camel Casing are used throughout this document.

Pascal Casing - First character of all words are Upper Case and other characters are lower case.

Example: BackColor

Camel Casing - First character of all words, except the first word are Upper Case and other characters are lower case.

Example: backColor

Consistency is the key to maintainable code. This statement is most true for naming your projects, source files, and identifiers including Fields, Variables, Properties, Methods, Parameters, Classes, Interfaces, and Namespaces.

2.1 General Guidelines

1. Always use Pascal Case name.
2. Avoid ALL CAPS and all lowercase name.
3. Always choose meaningful and specific names.
4. Avoid naming conflicts with existing .NET Framework namespaces or types.

2.2 Name Usage & Syntax

Identifier	Naming Convention
Project File	Pascal Case. Always match Assembly name and Root Namespace.
Source File	Pascal Case. Always match Class name and file name.
Resource of Embedded file	Try to use Pascal Case
Namespace	Pascal Case
Class/Struct	Pascal Case. Use a noun or noun phrase for class name.
Interface	Pascal Case. Always prefix interface name with capital “I”
Generic Class & Generic Parameter Type	Always use a single capital letter, such as T or K
Method	Pascal Case. Try to use a Verb or Verb-Object pair.
Property	Pascal Case. Property name should represent the entity it returns.
Field (public, protected, internal)	Pascal Case. Avoid using non-private Fields. Use Properties instead. Ex: public string Name; protected IList InnerList;
Field (private)	Camel Case and prefix with a single underscore (_) character. Ex: private string _name;
Delegate or Event	Treat as a Field

3 Coding Style

3.1 Instance members prefixed with this

This includes public, private and protected properties, fields, methods, and events.

```
public int Size { get; set; }

// Bad
Size = 55;

// Good
this.Size = 55;
```

3.2 Static members prefixed with the Class name

```
public static void DoSomething() { }

public void MyTask()
{
    // Bad
    DoSomething();

    // Good
    App.DoSomething();
}
```

3.3 Use { } for blocks (like if or for)

The curly braces should be on a separate line and not in the same line as if, for etc

```
// Bad
if (test) alert(message);

// Good
if (test)
{
     this.Notify(message);
}
```

3.4 No cryptic abbreviated parameter names

```
// Bad
public Rectangle(int wd, int ht) { }

// Good
public Rectangle(int width, int height) { }
```

3.5 Using Pascal Case for class names, method names, property names.

```
public class ClientActivity
{
    public void ClearStatistics()
    {
        //...
    }
    public void CalculateStatistics()
    {
        //...
```

```
    }
}
```

```
// Bad
private void doSomething() { }

// Good
private void DoSomething() { }
```

```
// Bad
public string name;
// Good
public string Name;
```

```
// Bad
public string name { get; set; }

// Good
public string Name { get; set; }
```

3.6 Using Camel Case for method arguments and local variables, member variables

```
public class UserLog
{
    public void Add(LogEvent logEvent)
    {
        int itemCount = logEvent.Items.Count;
        // ...
    }
}
```

```
// Bad
private string Name;

// Good
private string name;
```

```
// Bad
private string Name { get; set; }

// Good
private string name { get; set; }
```

3.7 Comments

- Using // or /// for comments, avoid using /* ... */
- Do not write comments for every line of code and every variable declared.
- Write comments wherever required. But good readable code will require very less comments. If all variables and method names are meaningful, that would make the code very readable and will not need many comments.

- Do not write comments if the code is easily understandable without comment. The drawback of having lot of comments is, if you change the code and forget to change the comment, it will lead to more confusion.
- Fewer lines of comments will make the code more elegant. But if the code is not clean/readable and there are less comments, that is worse.
- If you have to use some complex or weird logic for any reason, document it very well with sufficient comments.
- If you initialize a numeric variable to a special number other than 0, -1, etc, document the reason for choosing that value.
- The bottom line is, write clean, readable code such a way that it doesn't need any comments to understand.
- Perform spelling check on comments and also make sure proper grammar and punctuation is used.

3.8 Indentation and Spacing

- Use TAB for indentation. Do not use SPACES. Define the Tab size as 4.
- Use one blank line to separate logical groups of code

Block/Return on the new line. There should be one and only one single blank line between each method inside the class.

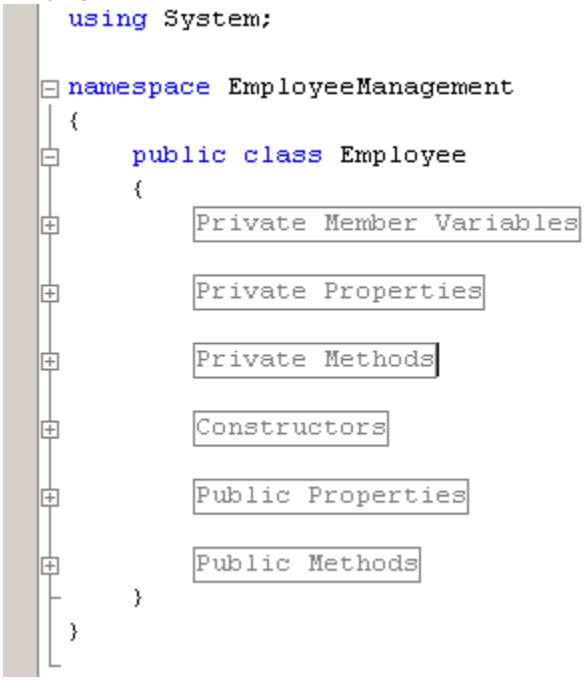
```
// Bad
var test = this.IsValid;
if (test)
{
}

// Good
var test = this.IsValid;

if (test)
{
}


// Bad
var test = this.IsValid;
while (test)
{
}


// Good
var test = this.IsValid;

while (test)
{
}



// Bad
public string GetMessage()
{
    var msg = "Hello";
    return msg;
}
```

```
// Good
public string GetMessage()
{
    var msg = "Hello";

    return msg;
}
```

- Use a single space before and after each operator and brackets.

```
// Good
if ( showResult == true )
{
    for ( int i = 0; i < 10; i++ )
    {
        //
    }
}

// Bad
if(showResult==true)
{
    for(int        i= 0;i<10;i++)
    {
        //
    }
}
```

- Use `#region` to group related pieces of code together. If you use proper grouping using `#region`, the page should like this when all definitions are collapsed.



3.9 Do use noun or noun phrases to name a class.

3.10 Prefix interfaces with the letter I. Interface names are noun (phrases) or adjectives.

```
public interface IShape
{
}
```

```
public interface IShapeCollection
{
}
public interface IGroupable
{
}
```

3.11 Name source files according to their main classes.

3.12 Organize namespaces with a clearly defined structure

```
// Examples
namespace Company.Product.Module.SubModule
namespace Product.Module.Component
namespace Product.Layer.Module.Group
```

3.13 Do declare all private member variables, properties and methods at the top of a class, with static variables at the very top and public members in the bottom.

```
// Correct
public class Account
{
    public static string BankName;
    public static decimal Reserves;

    public string Number {get; set;}
    public DateTime DateOpened {get; set;}
    public DateTime DateClosed {get; set;}
    public decimal Balance {get; set;}

    // Constructor
    public Account()
    {
        // ...
    }
}
```

3.14 Don't use **Hungarian** notation or any other type identification in identifiers

```
// Correct
int counter;
string name;

// Avoid
int iCounter;
string strName
```

3.15 Don't use ALL CAPs for constants or readonly variables

```
// Correct
public static const string ShippingType = "DropShip";

// Avoid
public static const string SHIPPINGTYPE = "DropShip";
```

3.16 Avoid using **Abbreviations**. Exceptions: abbreviations commonly used as names, such as Id, Xml, Ftp, Uri

```
// Correct
UserGroup userGroup;
Assignment employeeAssignment;

// Avoid
```

```
UserGroup usrGrp;
Assignment empAssignment;

// Exceptions
CustomerId customerId;
XmlDocument xmlDocument;
FtpHelper ftpHelper;
UriPart uriPart;
```

3.17 Using tools

R# - **ReSharper** or Visual Studio **IntelliSense**.

4 Checklist & Guideline

1. Make sure that there shouldn't be any **project warnings**.

2. It will be much better if ***Code Analysis*** is performed on a project (*with all Microsoft Rules enabled*) and then remove the warnings.

3. All **unused usings** need to be removed. Code cleanup for unnecessary code is always a good practice.

Refer: http://msdn.microsoft.com/en-us/magazine/ee335722.aspx.

4. '**null**' check need to be performed wherever applicable to avoid the *Null Reference Exception* at runtime.

5. Naming conventions to be followed always. Generally for variables/parameters follow **Camel casing** and for method names and class names follow **Pascal casing**.

Refer: http://msdn.microsoft.com/en-us/library/ms229043.aspx.

6. Make sure that you are aware of **SOLID** principles.

7. **Code Reusability**: Extract a method if same piece of code is being used more than once or you expect it to be used in future. Make some generic methods for repetitive task and put them in a related class so that other developers start using them once you intimate them. Develop user controls for common functionality so that they can be reused across the project.

Refer: http://msdn.microsoft.com/en-us/library/office/aa140806(v=office.10).aspx

http://blogs.msdn.com/b/frice/archive/2004/06/11/153709.aspx

8. **Code Consistency**: Let's say that an Int32 type is coded as int and String type is coded as string then they should be coded in that same fashion across the application. But not like sometimes int and sometimes as Int32.

```
int age;    (not Int16)
string name;  (not String)
object contactInfo; (not Object)
```

9. **Code Readability**: Should be maintained so that other developers understand your code easily.

10. **Disposing of Unmanaged Resources** like File I/O, Network resources, etc. They have to be disposed of once their usage is completed. Use usings block for unmanaged code if you want to automatically handle the disposing of objects once they are out of scope.

Refer: http://msdn.microsoft.com/en-us/library/498928w2.aspx

11. Proper implementation of **Exception Handling** (try/catch and finally blocks) and logging of exceptions.

Refer: http://msdn.microsoft.com/en-us/library/vstudio/ms229005(v=vs.100).aspx

12. Avoid writing very long methods. A method should typically have 1~25 lines of code. If a method has more than 25 lines of code, you must consider re factoring into separate methods.

13. Timely check-in/check-out of files/pages at source control (like TFS).

Refer: http://www.codeproject.com/Tips/593014/Steps-Check-in-Check-Out-Mechanism-for-TFS-Toavoi

14. **Peer code reviews**. Swap your code files/pages with your colleagues to perform internal code reviews.

15. **Unit Testing**. Write developer test cases and perform unit testing to make sure that basic level of testing is done before it goes to QA testing.

16. **Avoid nested for/foreach loops** and nested if conditions as much as possible.

17. Use **anonymous types** if code is going to be used only once.

Refer: http://msdn.microsoft.com/en-us/library/vstudio/bb397696.aspx

18. Try using **LINQ** queries and **Lambda** expressions to improve Readability.

Refer: http://msdn.microsoft.com/en-us/library/bb308959.aspx

19. Proper usage of var, object, and dynamic keywords. They have some similarities due to which most of the developers have confusions or don't know much about them and hence they use them interchangeably, which shouldn't be the case.

Refer: https://blogs.msdn.microsoft.com/csharpfaq/2010/01/25/what-is-the-difference-between-dynamic-and-object-keywords/

20. Use **access specifiers** (*private, public, protected, internal, protected internal*) as per the scope need of a method, a class, or a variable. Let's say if a class is meant to be used only within the assembly then it is enough to mark the class as internal only.

Refer: http://msdn.microsoft.com/en-us/library/kktasw36.aspx

21. Use interfaces wherever needed to maintain decoupling. Some design patterns came into existence due to the usage of interfaces.

Refer: http://msdn.microsoft.com/en-IN/library/3b5b8ezk(v=vs.100).aspx

22. Mark a class as **sealed** or **static** or **abstract** as per its usage and your need.

Refer: http://msdn.microsoft.com/en-us/library/ms173150(v=vs.100).aspx

23. Use a StringBuilder instead of string if multiple concatenations required, to save **heap memory**.

24. Check whether any **unreachable code** exists and modify the code if it exists.

25. Write **comments** on top of all methods to describe their usage and expected input types and return type information.

26. Use **fiddler** tool to check the HTTP/network traffic and bandwidth information to trace the performance of web application and services.

27. Use *WCFTestClient.exe* tool if you want to verify the service methods out of the visual studio or by attaching its process to visual studio for debugging purpose.

28. Use **constants** and **readonly** wherever applicable.

Refer:

http://msdn.microsoft.com/en-us/library/acdd6hb7(v=vs.100).aspx

http://msdn.microsoft.com/en-us/library/e6w8fe1b(v=vs.100).aspx

29. Avoid **type casting and type conversions** as much as possible; because it is a performance penalty.

Refer: http://msdn.microsoft.com/en-us/library/ms173105.aspx

30. Override ToString (from Object class) method for the types which you want to provide with custom information.

Refer: http://msdn.microsoft.com/en-us/library/ms173154(v=vs.100).aspx

31. Avoid straightaway **copy/pasting of code** from other sources. It is always recommended to hand written the code even though if you are referring the code from some sources. By this you will get good practice of writing yourself the code and also you will understand the proper usage of that code; finally you never forget it.

32. Always make it a practice to read **books/articles**, upgrade and follow the Best Practices and Guidelines by industry experts like **Microsoft experts** and well-known authors like Martin Fowler, Kent Beck, Jeffrey Ritcher, Ward Cunningham, Scott Hanselman, Scott Guthrie, Donald E Knuth.

33. Verify whether your code have any **memory leakages**. If yes, make sure that have been fixed.

Refer: http://blogs.msdn.com/b/davidklinems/archive/2005/11/16/493580.aspx

34. Try attending **technical seminars by experts** to be in touch with the latest software trends and technologies and best practices.

35. Understand thoroughly the **OOPs concepts** and try implementing it in your code.

36. Get to know about your **project design and architecture** to better understand the *flow of your application* as a whole.

37. Take necessary steps to block and avoid any **cross scripting attacks, SQL injection**, and other security holes.

38. Always **encrypt** (by using good **encryption algorithms**) secret/sensitive information like passwords while saving to database and connection strings stored in *web.config* file(s) to avoid manipulation by unauthorized users.

39. Avoid using **default** keyword for the known types (primitive types) like int, decimal, bool, etc. Most of the times it should be used in case of Generic types (T) as we may not be sure whether the type is a value type or reference type.

Do use predefined type names instead of system type names like Int16, Single, UInt64, etc

```
// Correct
string firstName;
int lastIndex;
bool isSaved;

// Avoid
String firstName;
Int32 lastIndex;
Boolean isSaved;
```

Refer: http://msdn.microsoft.com/en-us/library/xwthohod(v=vs.100).aspx

40. Usage of '`out`' and '`ref`' keywords be avoided as recommended by Microsoft (*in the Code analysis Rules and guidelines*). These keywords are used to pass parameters by reference. Note that '`ref`' parameter should be initialized in the calling method before passing to the called method but for '`out`' parameter this is not mandatory.

41. Use String.Empty instead of “”

Good:

```
if ( name == String.Empty )
{
        // do something
}
```

Not Good:

```
if ( name == “” )
{
        // do something
}
```

42. Avoid having very large files. If a single file has more than 1000 lines of code, it is a good candidate for refactoring. Split them logically into two or more classes.

43. Avoid passing too many parameters to a method. If you have more than 4~5 parameters, it is a good candidate to define a class or structure.

5 Architecture

1. Always use multi layer (N-Tier) architecture.
2. Never access database from the UI pages. Always have a data layer class which performs all the database related tasks. This will help you support or migrate to another database back end easily.
3. Use try-catch in your data layer to catch all database exceptions. This exception handler should record all exceptions from the database. The details recorded should include the name of the command being executed, stored proc name, parameters, connection string used etc. After recording the exception, it could be re thrown so that another layer in the application can catch it and take appropriate action.

4. Separate your application into multiple assemblies. Group all independent utility classes into a separate class library. All your database related files can be in another class library.